Лабораторная работа 2
Сортировки

Финоченко Александр Викторович Б02-201

Цель работы: написать улучшения к сортировкам с квадратичной асимптотической сложностью, провести декомпозицию на элементарные шаги и протестировать отдельные функции.

Шейкерская сортировка

Во всех следующих программах использовались обозначения

```
using U = unsigned;
using ULL = unsigned long long;
```

Ниже приведены функции прямого прохода, обратного прохода и шейкерской сортировки.

```
bool forward_step(U arr[], U const begin_idx, U const end_idx) {
    bool sorted = true;
    for (U idx = begin_idx; idx != end_idx; ++idx) {
        if (arr[idx] > arr[idx + 1]) {
            auto tmp = arr[idx];
            arr[idx] = arr[idx + 1];
            arr[idx + 1] = tmp;
            sorted = false;
        }
    }
    return sorted;
}

bool backward_step(U arr[], U const begin_idx, U const end_idx) {
    bool sorted = true;
    for (U idx = end_idx - 1; idx != begin_idx - 1; --idx) {
        if (arr[idx] > arr[idx + 1]) {
            auto tmp = arr[idx];
            arr[idx] = arr[idx + 1];
            arr[idx + 1] = tmp;
            sorted = false;
        }
    }
    return sorted;
}

void shaker_sort(U arr[], U const begin_idx, U const end_idx) {
    bool sorted = false;
    while (!sorted) {
        sorted = forward_step(arr, begin_idx, end_idx);
        sorted = backward_step(arr, begin_idx, end_idx);
    }
    assert(is_sorted(arr, begin_idx, end_idx));
}
```

Проверка того, что массив отсортирован, производилась с помощью функции is_sorted:

```
template <typename T>
bool is_sorted(T arr[], U const begin_idx, U const end_idx) {
    bool sorted = true;
    for (U idx = begin_idx; idx != end_idx; idx++) {
        if (arr[idx] > arr[idx + 1])
            sorted = false;
    }
    return sorted;
}
```

Программа успешно отработала, что означает, что функция сортировки работает нормально.

Сортировка расчёской

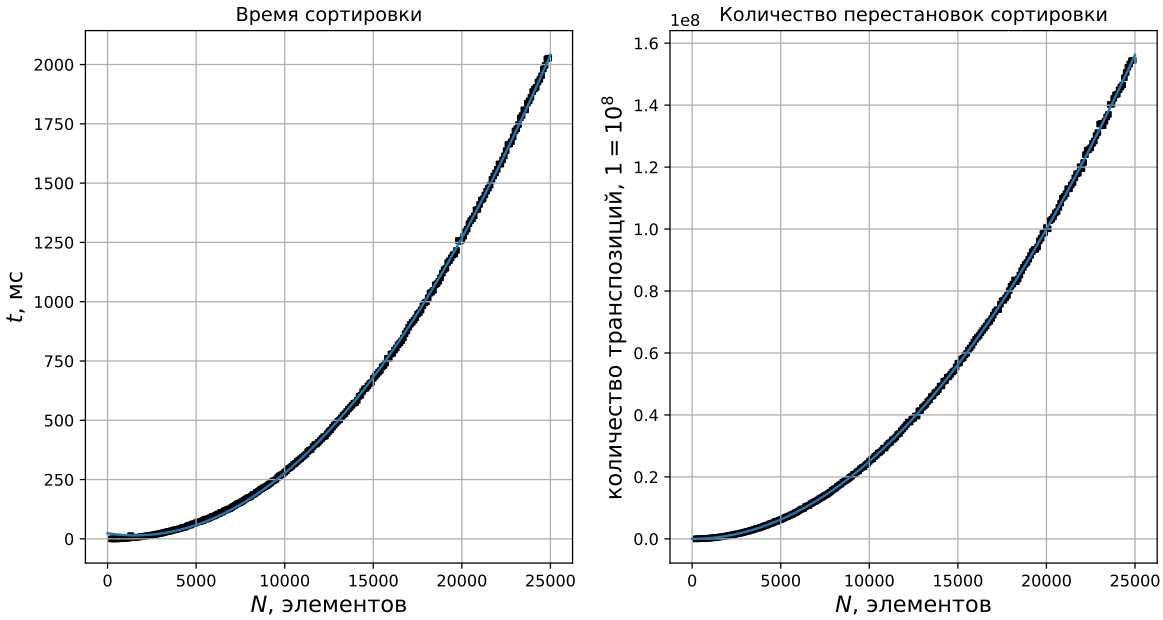
Сортировка проводится сначала по $N/4$ и $N/2$, далее проводится обычная сортировка пузырьком. Полный код функции сортировки расчёской представлен ниже

```
template <typename T>
bool is_N4_sorted(T arr[], U const begin_idx, U const end_idx) {
    U step = (end_idx - begin_idx + 1) / 4;
    bool sorted = true;
    for (U idx = begin_idx; idx + step <= end_idx; idx += step) {
        if (arr[idx] > arr[idx + step])
            sorted = false;
    }
    return sorted;
}

template <typename T>
bool is_N2_sorted(T arr[], U const begin_idx, U const end_idx) {
    U step = (end_idx - begin_idx + 1) / 2;
    bool sorted = true;
    for (U idx = begin_idx; idx + step <= end_idx; idx += step) {
        if (arr[idx] > arr[idx + step])
            sorted = false;
    }
    return sorted;
}

ULL N4_sort(U arr[], U const begin_idx, U const end_idx) {
    U step = (end_idx - begin_idx + 1) / 4;
    ULL count = 0;
    bool sorted = false;
    while (!sorted) {
        sorted = true;
        for (U idx = begin_idx; idx + step <= end_idx; idx += step) {
            if (arr[idx] > arr[idx + step]) {
                auto tmp = arr[idx];
                arr[idx] = arr[idx + step];
                arr[idx + step] = tmp;
                sorted = false;
                count++;
            }
        }
    }
    assert(is_N4_sorted(arr, begin_idx, end_idx));
    return count;
}
```

```
42
43 ULL N2_sort(U arr[], U const begin_idx, U const end_idx) {
44     U step = (end_idx - begin_idx + 1) / 2;
45     ULL count = 0;
46     bool sorted = false;
47     while (!sorted) {
48         sorted = true;
49         for (U idx = begin_idx; idx + step <= end_idx; idx += step) {
50             if (arr[idx] > arr[idx + step]) {
51                 auto tmp = arr[idx];
52                 arr[idx] = arr[idx + step];
53                 arr[idx + step] = tmp;
54                 sorted = false;
55                 count++;
56             }
57         }
58     }
59     assert(is_N2_sorted(arr, begin_idx, end_idx));
60     return count;
61 }
62
63 ULL comb_sort(U arr[], U const begin_idx, U const end_idx) {
64     ULL count1 = N4_sort(arr, begin_idx, end_idx);
65     ULL count2 = N2_sort(arr, begin_idx, end_idx);
66     ULL count = 0;
67
68     bool sorted = false;
69     while (!sorted) {
70         sorted = true;
71         for (U idx = begin_idx; idx != end_idx; ++idx) {
72             if (arr[idx] > arr[idx + 1]) {
73                 auto tmp = arr[idx];
74                 arr[idx] = arr[idx + 1];
75                 arr[idx + 1] = tmp;
76                 sorted = false;
77                 count++;
78             }
79         }
80     }
81     return count1 + count2 + count;
82 }
```

Программа была проверена на случайно сгенерированных массивах и ошибок не выдавала. Сама функция одновременно сортирует массив и подсчитывает количество перестановок.

Далее представлен график зависимости времени и числа перестановок в зависимости от количества элементов N



Видно, что зависимость квадратичная.

Сортировка Шелла

Общая функция для сортировки Шелла с некоторым шагом

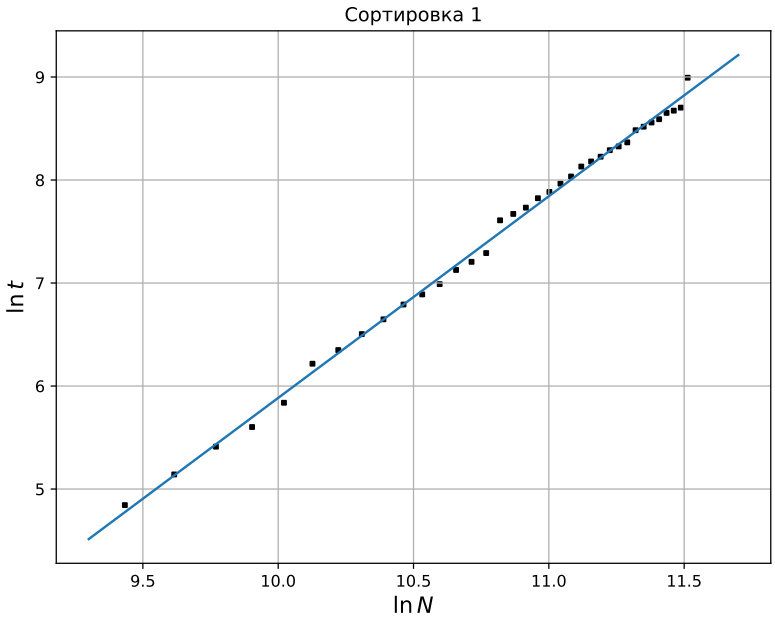
```
ULL shell_sort_i(U arr[], U const begin_idx, U const end_idx, U step) {
    ULL count = 0;
    if (begin_idx + step > end_idx)
        return 0;
    for (int last = end_idx - step; last >= 0; last -= step) {
        auto tmp_idx = last;
        while (tmp_idx + step <= end_idx && arr[tmp_idx] > arr[tmp_idx + step]) {
            auto tmp = arr[tmp_idx];
            arr[tmp_idx] = arr[tmp_idx + step];
            arr[tmp_idx + step] = tmp;
            tmp_idx += step;
            count++;
        }
    }
    return count;
}
```

Последовательность $d_{i+1} = d_i/2, d_1 = N$

Для шага $d_{i+1} = d_i/2, d_1 = N$ функция сортировки Шелла выглядит так

```
ULL shell_sort(U arr[], U const begin_idx, U const end_idx) {
    ULL count = 0;
    U step = (end_idx - begin_idx + 1);

    while (step != 0) {
        count += shell_sort_i(arr, begin_idx, end_idx, step);
        step /= 2;
    }
    return count;
}
```

Предполагая, что при больших N время работы программы приближается к $C \cdot N^{\alpha}$, определим это α. Для этого перейдём к логарифмам, тогда $\ln t = \alpha \ln N + \ln C$. Получаем, что $\alpha = 1.96$



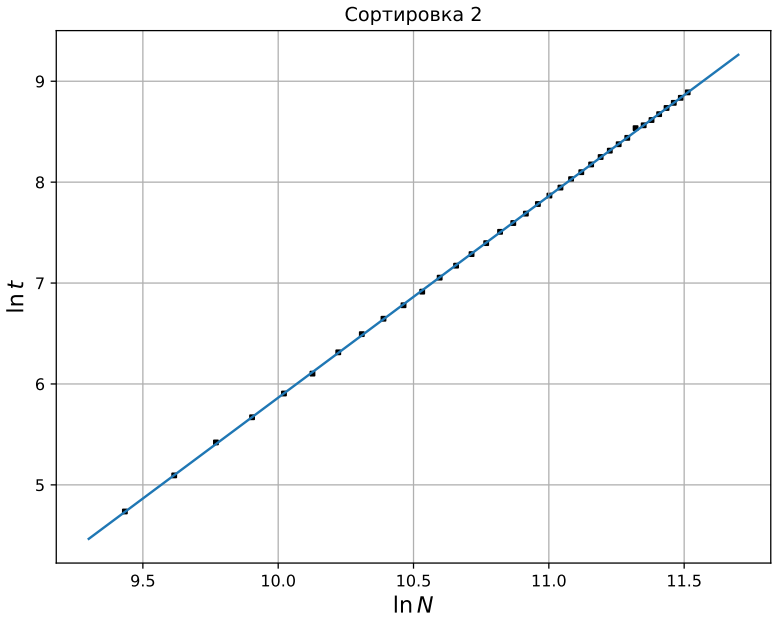
Среднее количество перестановок равно $6.88 \cdot 10^8$.

Последовательности Хиббарда

Проделаем тоже самое. Функция функция сортировки Шелла выглядит так

```
ULL shell_sort(U arr[], U const begin_idx, U const end_idx) {
    ULL count = 0;
    U i = 0;
    while ((U)pow(2, i) - 1 <= (end_idx - begin_idx + 1)) {
        ++i;
    }
    --i;
    U step = (U) pow(2, i) - 1;
    while (step != 0) {
        count += shell_sort_i(arr, begin_idx, end_idx, step);
        i--;
        step = pow(2, i) - 1;
    }
    return count;
}
```

Если $\ln t = \alpha \ln N + \ln C$, то $\alpha = 1.99$



Среднее количество перестановок равно $7.06 \cdot 10^8$.

Обратная последовательность Фибоначчи

Для обратной последовательности Фибоначчи функция сортировки Шелла выглядит так

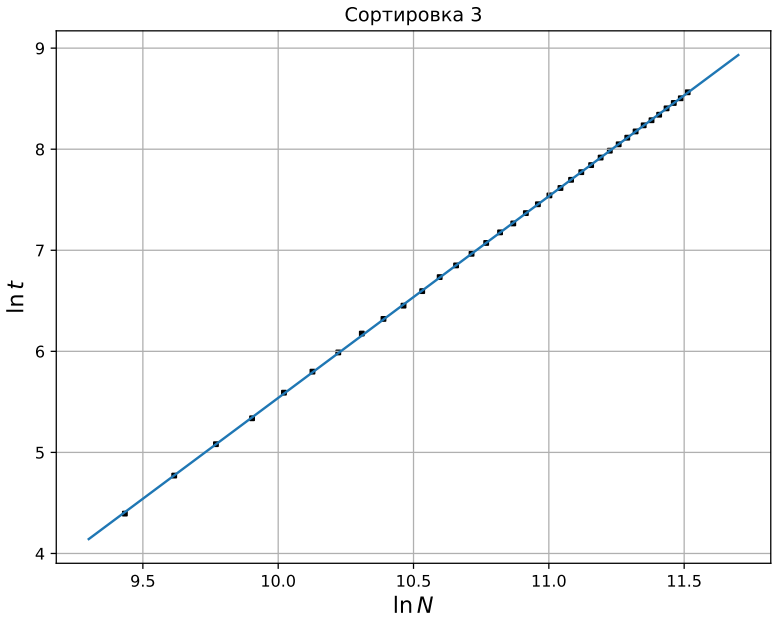
```
ULL shell_sort(U arr[], U const begin_idx, U const end_idx) {
    ULL count = 0;
    vector<U> f_arr = fib_arr(end_idx - begin_idx + 1);
    U step;
    for (U i = size(f_arr) - 1; i != 1; i--) {
        step = f_arr[i];
        count += shell_sort_i(arr, begin_idx, end_idx, step);
    }
    return count;
}
```

Здесь проще использовать векторы. Функция fib_arr нужна, чтобы быстро находить числа Фибоначчи

```
vector<U> fib_arr(U N) {
    U F_0 = 0; U F_1 = 1;
    U n = 0;
    while (F_1 <= N) {
        U temp = F_1;
        F_1 += F_0;
        F_0 = temp;
        n++;
    }
    F_0 = 0; F_1 = 1;
    vector<U> fib_arr(n + 1);
    fib_arr[0] = 0;
    for (U i = 1; i <= n; i++) {
        fib_arr[i] = F_1;
        U temp = F_1;
        F_1 += F_0;
        F_0 = temp;
    }
```

```
19      return fib_arr;
20  }
```

Если $\ln t = \alpha \ln N + \ln C$, то $\alpha = 1.99$



Среднее количество перестановок равно $5.07 \cdot 10^8$.

Вывод

Получаем, что наиболее эффективная по времени является 1 последовательность, наиболее эффективная по числу перестановок - 3 последовательность.